2. 클래스와 객체

인스턴스의 문자열 표현식 변형

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)
    def __str__(self):
        return '({0.x}, {0.y})'.format(self)

p = Pair(3, 4)
p
```

```
Pair(3, 4)
```

"인스턴스의 문자열 표현식을 바꾸려면 __str__()와 __repr__() 메소드를 정의 한다."

인스턴스의 문자열 표현식 변형

```
print(p)
```

```
(3, 4)
```

"객체를 출력할 경우 기본적으로는 __str__()함수가 호출 된다."

```
print('p is {0!r}'.format(p))
```

```
p is Pair(3, 4)
```

"!r은 기본 값으로 __repr__()함수가 호출 된다."

```
print('p is {0}'.format(p))
```

```
p is (3, 4)
```

"{}은 기본 값으로 __str__()함수가 호출 된다."

"__repr__()는 eval(repr(x)) == x와 같은 텍스트를 만드는 것이 표준이다."

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

객체의 콘텍스트 관리 프로토콜 지원

```
if __name__ == '__main__':
    from functools import partial

    c = LazyConnection(('www.python.org', 80))
    with c as s:
        s.send(b'GET /index.html HTTP/1.0\r\n')
        s.send(b'Host: www.python.org\r\n')
        s.send(b'\r\n')
        resp = b''.join(iter(partial(s.recv, 8192), b''))

    print('Got %d bytes' % len(resp))
```

```
Got 392 bytes
```

"객체가 콘텍스트 관리 프로토콜(with 구문)을 지원하게 구현"

with문을 만나면 __enter__() 메소드가 호출된다.
with문을 빠져 나갈때는 __exit__() 메소드가 호출된다.

인스턴스를 많이 생성할 때 메모리 절약

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

d1 = Date(2020, 6, 1)
d1.__dict__
```

```
{'year': 2020, 'month': 6, 'day': 1}
```

"객체 인스턴스 마다 내부에 멤버를 저장할 목적으로 딕셔너리를 구성한다. 이는 메모리 낭비의 원인이 된다."

인스턴스를 많이 생성할 때 메모리 절약

```
class Date:
    __slots__ = ['year', 'month', 'day']
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

d1 = Date(2020, 6, 1)
dir(d1)
```

```
['__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 ...
```

"__slots__을 정의하면 파이썬은 인스턴스에 훨씬 더 압축된 내부 표현 식을 사용한다.
인트턴스 마다 딕셔너리를 구성하지 않고 튜플이나 리스트 값은 부피가 작은 고정 배열로 인스턴스가 만들어 진다. "

관리 속성 만들기

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value
```

"인스턴스 속성을 얻거나 설정할 때 추가적인 처리(타입 체크, 검증 등)를 하고 싶다."

```
if __name__ == '__main__':
    a = Person('Guido')
    print(a.first_name)
    a.first_name = 'Dave'
    print(a.first_name)
    try:
        a.first_name = 42
    except TypeError as e:
        print(e)
```

```
Guido
Dave
Expected a string
```

"@first_name.setter로 데코레이터를 지정하면 해당 속성을 변경하려 할 때 자동으로 호출 된다.

이때 타입을 검사할 수 있고 문자열이 아닌경우 예외를 던진다."

2. 클래스와 객체

2.1 클래스와 객체 I

2.2 클래스와 객체 II

2.3 클래스와 객체 III

부모 클래스의 메소드 호출


```
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def spam(self):
        print('B.spam')
        super().spam()

if __name__ == '__main__':
    b = B()
    b.spam()
```

```
B.spam
A.spam
```

"부모의 메소드를 호출하려면 super()함수를 사용한다."

부모 클래스의 메소드 호출

```
class A:
    def __init__(self):
        self.x = 0

class B(A):
    def __init__(self):
        super().__init__()
        self.y = 1

if __name__ == '__main__':
    b = B()
    print(b.x, b.y)
```

```
0 1
```

"super()는 일반적으로 __init__() 메소드에서 부모를 제대로 초기화하기 위해 사용한다.

```
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    def __getattr__(self, name):
        return getattr(self._obj, name)

    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)
        else:
            setattr(self._obj, name, value)
```

"파이썬의 특별 메소드를 오버라이드한 코드에서 super()를 사용 하기도 한다.
__setattr__() 구현에 이름 확인이 들어 있다. 만약 이름이 밑줄로 시작하면 super()를 사용해서 __setattr__()의 원래의 구현을 호출한다."

부모 클래스의 메소드 호출

```
if __name__ == '__main__':
    class A:
        def __init__(self, x):
            self.x = x
        def spam(self):
            print('A.spam')

    a = A(42)
    p = Proxy(a)
    print(p.x)
    print(p.spam())
    p.x = 37
    print('Should be 37:', p.x)
    print('Should be 37:', a.x)
```

```
42
A.spam
None
Should be 37: 37
Should be 37: 37
```

부모 클래스의 메소드 호출


```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')

if __name__ == '__main__':
    c = C()
```

부모 클래스의 메소드 호출

```
Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__
```

"이 코드를 실행 하면 Base.__init__() 메소드가 두 번 호출된다."

부모 클래스의 메소드 호출


```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A,B):
    def __init__(self):
        super().__init__()
        print('C.__init__')

if __name__ == '__main__':
```

부모 클래스의 메소드 호출

```
Base.__init__
B.__init__
A.__init__
C.__init__
```

"super()를 사용하여 코드를 수정한다면 올바르게 동작한다."

서브클래스에서 프로퍼티 확장


```
class Person:
    def __init__(self, name):
        self.name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._name = value

    @name.deleter
    def name(self):
        raise AttributeError("Can't delete attribute")
```

서브클래스에서 프로퍼티 확장

```
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

"서브 클래스에서, 부모 클래스에 정의한 프로퍼티의 기능을 확장하고 싶다."

서브클래스에서 프로퍼티 확장

```
if __name__ == '__main__':
    a = SubPerson('Guido')
    print(a.name)
    a.name = 'Dave'
    print(a.name)
    try:
        a.name = 42
    except TypeError as e:
        print(e)
```

```
Setting name to Guido
Getting name
Guido
Setting name to Dave
Getting name
Dave
Setting name to 42
Expected a string
```

"부모의 메소드에 도달하기 위한 유일한 방법은 인스턴스 변수가 아닌 클래스 변수로 접근해야 한다. super(SubPerson, SubPerson)으로 구현한다."

새로운 클래스나 인스턴스 속성 만들기

```
class Integer:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]
```

"타입 확인 등과 같이 추가적 기능을 가진 해로운 종류의 인스턴스 속성을 만들고 싶다."

새로운 클래스나 인스턴스 속성 만들기

```
class Point:
    x = Integer('x')
    y = Integer('y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

if __name__ == '__main__':
    p = Point(2, 3)
    print(p.x)
    p.y = 5
    try:
        p.x = 2.3
    except TypeError as e:
        print(e)
```

```
2
Expected an int
```

"완전히 새로운 종류의 인스턴스 속성을 만들려면, 그 기능을 디스크립터 클래스 형태로 정의해야 한다."

게으른 계산을 하는 프로퍼티 사용

```
class lazyproperty:
    def __init__(self, func):
        self.func = func
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            value = self.func(instance)
            setattr(instance, self.func.__name__, value)
            return value
```

"읽기 전용 속성을 프로퍼티로 정의하고, 이 속성에 접근할 때만 계산하도록 하고 싶다. 하지만 한 번 접근하고 나면 이 값을 캐시해 놓고 다음 번에 접근할 때에는 다시 계산하지 않도록 하고 싶다."

게으른 계산을 하는 프로퍼티 사용

```
if __name__ == '__main__':
    import math
    class Circle:
        def __init__(self, radius):
            self.radius = radius

        @lazyproperty
        def area(self):
            print('Computing area')
            return math.pi * self.radius ** 2

        @lazyproperty
        def perimeter(self):
            print('Computing perimeter')
            return 2 * math.pi * self.radius

c = Circle(4.0)
c.radius
```

```
4.0
```

```
c.area
```

```
Computing area
50.26548245743669
```

```
c.area
```

```
50.26548245743669
```

```
c.perimeter
```

```
Computing perimeter
25.132741228718345
```

```
c.perimeter
```

```
25.132741228718345
```

"Computing area"와 "Computing perimeter" 메시지가 한번만 나타나는 점에 주목하자.

자료 구조 초기화 단순화하기

```
class Structure:
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        for name, value in zip(self._fields, args):
            setattr(self, name, value)

if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    class Point(Structure):
        _fields = ['x','y']

    class Circle(Structure):
        _fields = ['radius']
        def area(self):
            return math.pi * self.radius ** 2
```

"자료 구조로 사용하는 클래스를 작성하고 있는데, 반복적으로 비슷한 __init__()함수를 매번 작성해야 한다."

```
if __name__ == '__main__':
    s = Stock('ACME', 50, 91.1)
    print(s.name, s.shares, s.price)

    p = Point(2,3)
    print(p.x, p.y)

    c = Circle(4.5)
    print(c.radius)

    try:
        s2 = Stock('ACME', 50)
    except TypeError as e:
        print(e)
```

```
ACME 50 91.1
2 3
4.5
Expected 3 arguments
```

자료 구조 초기화 단순화하기

```
class Structure:
    _fields= []
    def __init__(self, *args, **kwargs):
        if len(args) > len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        for name in self._fields[len(args):]:
            setattr(self, name, kwargs.pop(name))

        if kwargs:
            raise TypeError('Invalid argument(s): {}'.format(','.join(kwargs)))
```

```
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, price=91.1)
    s3 = Stock('ACME', shares=50, price=91.1)
```

"키워드 매개변수를 지원하기로 결정 했다면 사용할 수 있는 디자인 옵션이 몇 가지 있다. 그중 한가지는 키워드 매개변수를 매핑해서 _fields에 명시된 속성 이름에만 일치하도록 만드는 것이다."

자료 구조 초기화 단순화하기

```
class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args, **kwargs):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the additional arguments (if any)
        extra_args = kwargs.keys() - self._fields
        for name in extra_args:
            setattr(self, name, kwargs.pop(name))
        if kwargs:
            raise TypeError('Duplicate values for {}'.format(','.join(kwargs)))
```

자료 구조 초기화 단순화하기

```
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, 91.1, date='6/1/2020')
```

"_fields에 명시되지 않은 구조에 추가적인 속성을 추가하는 수단으로 키워드 매개변수를 사용할 수 있다."

2. 클래스와 객체

인터페이스, 추상 베이스 클래스 정의

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxbytes=-1):
        pass
    @abstractmethod
    def write(self, data):
        pass

class SocketStream(IStream):
    def read(self, maxbytes=-1):
        print('reading')
    def write(self, data):
        print('writing')

def serialize(obj, stream):
    if not isinstance(stream, IStream):
        raise TypeError('Expected an IStream')
    print('serializing')
```

```
if __name__ == '__main__':
    try:
        a = IStream()
    except TypeError as e:
        print(e)

    a = SocketStream()
    a.read()
    a.write('data')

    serialize(None, a)

    import sys
    try:
        serialize(None, sys.stdout)
    except TypeError as e:
        print(e)

    import io
    IStream.register(io.IOBase)

    serialize(None, sys.stdout)
```

인터페이스, 추상 베이스 클래스 정의

```
Can't instantiate abstract class IStream with abstract methods read,
write
reading
writing
serializing
Expected an IStream
serializing
```

"인터페이스나 추상 베이스 클래스 역할을 하는 클래스를 정의 하고 싶다. 그리고 이 클래스는 타입 확인을 하고 특정 메소드가 서브 클래스에 구현되었는지 보장한다.

인터페이스, 추상 베이스 클래스 정의


```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @property
    @abstractmethod
    def name(self):
        pass

    @name.setter
    @abstractmethod
    def name(self, value):
        pass

    @classmethod
    @abstractmethod
    def method1(cls):
        pass

    @staticmethod
    @abstractmethod
    def method2():
        pass
```

"@abstractmethod를 스태틱 메소드, 클래스 메소드, 프로퍼티에도 적용할 수 있다.

다만 함수 정의 직전에 @abstractmethod를 적용해야 한다.

커스텀 컨테이너 구현


```
import collections.abc
import bisect

class SortedItems(collections.Sequence):
    def __init__(self, initial=None):
        self._items = sorted(initial) if initial is not None else []

    def __getitem__(self, index):
        return self._items[index]

    def __len__(self):
        return len(self._items)

    def add(self, item):
        bisect.insort(self._items, item)
```

"리스트나 딕셔너리와 같은 내장 컨테이너와 비슷하게 동작하는 커스텀 클래스를 구현하고 싶다."

```
if __name__ == '__main__':
    items = SortedItems([5, 1, 3])
    print(list(items))
    print(items[0])
    print(items[-1])
    items.add(2)
    print(list(items))
    items.add(-10)
    print(list(items))
    print(items[1:4])
    print(3 in items)
    print(len(items))
    for n in items:
        print(n)
```

```
[1, 3, 5]
1
5
[1, 2, 3, 5]
[-10, 1, 2, 3, 5]
[1, 2, 3]
True
5
-10
1
2
3
5
```

"SortedItems의 인스턴스는 보통의 시퀀스와 동일한 동작을 하고, 인덱싱, 순환, len(), in 연산자, 자르기 등 일반적인 연산을 모두 지원한다."

속성 접근 델리게이팅

```
class A:
    def spam(self, x):
        print('A.spam')

    def foo(self):
        print('A.foo')

class B:
    def __init__(self):
        self._a = A()

    def bar(self):
        print('B.bar')

    def __getattr__(self, name):
        return getattr(self._a, name)
```

"인스턴스가 속성에 대한 접근을 내부 인스턴스로 델리게이트해서 상속의 대안으로 사용하거나 프록시 구현을 하고 싶다."

속성 접근 델리게이팅

```
if __name__ == '__main__':
    b = B()
    b.bar()
    b.spam(42)
```

```
B.bar
A.spam
```

"__getattr__() 메소드는 속성을 찾아 보는 도구 모음 정도로 생각하면 된다. 이 메소드는 코드가 존재하지 않는 속성에 접근하려 할 때 호출된다. 앞에 나온 코드에서 정의하지 않은 B에 대한 접근을 A로 델리케이트 한다."

속성 접근 델리게이팅

```
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    def __getattr__(self, name):
        print('getattr:', name)
        return getattr(self._obj, name)

    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)
        else:
            print('setattr:', name, value)
            setattr(self._obj, name, value)

    def __delattr__(self, name):
        if name.startswith('_'):
            super().__delattr__(name)
        else:
            print('delattr:', name)
            delattr(self._obj, name)
```

속성 접근 델리게이팅

```
if __name__ == '__main__':
    class Spam:
        def __init__(self, x):
            self.x = x
        def bar(self, y):
            print('Spam.bar:', self.x, y)

    s = Spam(2)

    p = Proxy(s)

    print(p.x)
    p.bar(3)
    p.x = 37
```

```
getattr: x
2
getattr: bar
Spam.bar: 2 3
setattr: x 37
```

"델리게이트의 또 다른 예제로 프록시 구현이 있다. 이 프록시 클래스를 사용하려면, 단순히 다른 인스턴스를 감싸면 된다."

속성 접근 델리게이팅

```
class ListLike:
    def __init__(self):
        self._items = []
    def __getattr__(self, name):
        return getattr(self._items, name)

    def __len__(self):
        return len(self._items)
    def __getitem__(self, index):
        return self._items[index]
    def __setitem__(self, index, value):
        self._items[index] = value
    def __delitem__(self, index):
        del self._items[index]
```

속성 접근 델리게이팅

```
if __name__ == '__main__':
    a = ListLike()
    a.append(2)
    a.insert(0, 1)
    a.sort()
    print(len(a))
    print(a[0])
```

```
2
1
```

"ListLike 객체를 만들려고 하면, append()와 insert() 같은 일반적인 리스트 메소드를 지원한다. 하지만 len(), 아이템 검색 등의 연산은 지원하지 않는다.

서로 다른 연산을 지원하려면, 수동으로 관련된 특별 메소드를 델리게이트 해야 한다."

클래스에 생성자 여러 개 정의

```
import time

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        t = time.localtime()
        return cls(t.tm_year, t.tm_mon, t.tm_mday)
```

"클래스를 작성 중인데, 사용자가 __init__()이 제공하는 방식 이외에 여러 가지 방식으로 인스턴스를 생성할 수 있도록 하고 싶다."

클래스에 생성자 여러 개 정의


```
if __name__ == '__main__':
    a = Date(2020, 6, 1)
    b = Date.today()
    print(a.year, a.month, a.day)
    print(b.year, b.month, b.day)

    class NewDate(Date):
        pass

    c = Date.today()
    d = NewDate.today()
    print('Should be Date instance:', Date)
    print('Should be NewDate instance:', NewDate)
```

```
2020 6 1
2020 5 30
Should be Date instance: <class '__main__.Date'>
Should be NewDate instance: <class '__main__.NewDate'>
```

"생성자를 여러 개 정의하려면 클래스 메소드를 사용해야 한다."

init 호출 없이 인스턴스 생성

```
from time import localtime

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        d = cls.__new__(cls)
        t = localtime()
        d.year = t.tm_year
        d.month = t.tm_mon
        d.day = t.tm_mday
        return d
```

"인스턴스를 생성해야 하는데, __init__() 메소드 호출을 피하고 싶다."

init 호출 없이 인스턴스 생성

```
d = Date.__new__(Date)
print(d)
print(hasattr(d,'year'))

data = {
    'year' : 2020,
    'month' : 6,
    'day' : 1
}

d.__dict__.update(data)
print(d.year, d.month)

d = Date.today()
print(d.year, d.month, d.day)
```

```
<__main__.Date object at 0x0000017824404D48>
False
2020 6
2020 5 30
```

"클래스의 __new__() 메소드를 호출해서 초기화 하지 않은 인스턴스를 생성할 수 있다."

```
import weakref

class Node:
    def __init__(self, value):
        self.value = value
        self._parent = None
        self.children = []

    def __repr__(self):
        return 'Node({!r:})'.format(self.value)

    @property
    def parent(self):
        return self._parent if self._parent is None else self._parent()

    @parent.setter
    def parent(self, node):
        self._parent = weakref.ref(node)

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

```
if __name__ == '__main__':
    root = Node('parent')
    c1 = Node('c1')
    c2 = Node('c2')
    root.add_child(c1)
    root.add_child(c2)

    print(c1.parent)
    del root
    print(c1.parent)
```

```
Node('parent')
None
```

"순환이 있는 자료 구조를 생성하는 프로그램에서 메모리 관리의 문제가 있다."

"상호 참조가 있는 구조에서는 weakref 라이브러리를 사용하는 약한 참조로 만드는 것을 고려해야 한다."

순환 자료 구조에서 메모리 관리

```
class Data:
    def __del__(self):
        print('Data.__del__')

class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

```
if __name__ == '__main__':
    a = Data()
    del a
    a = Node()
    del a
    a = Node()
    a.add_child(Node())
    del a
```

```
Data.__del__
Data.__del__
```

"부모노드와 자식노드가 상호 참조하는 경우 노드가 삭제되지 않는다."

```
import gc
gc.collect()
```

```
Data.__del__
Data.__del__
```

"가비지 컬렉터를 강제로 실행하면 메모리가 해지 된다. 하지만 가비지 컬렉터가 언제 동작할 지는 알수가 없다."

비교 연산을 지원하는 클래스 만들기

```
from functools import total_ordering
class Room:
    def __init__(self, name, length, width):
        self.name = name
        self.length = length
        self.width = width
        self.square_feet = self.length * self.width

@total_ordering
class House:
    def __init__(self, name, style):
        self.name = name
        self.style = style
        self.rooms = list()
```

비교 연산을 지원하는 클래스 만들기


```
@property
    def living_space_footage(self):
        return sum(r.square_feet for r in self.rooms)

    def add_room(self, room):
        self.rooms.append(room)

    def __str__(self):
        return '{}: {} square foot {}'.format(self.name,
                                              self.living_space_footage,
                                              self.style)

    def __eq__(self, other):
        return self.living_space_footage == other.living_space_footage

    def __lt__(self, other):
        return self.living_space_footage < other.living_space_footage
```

비교 연산을 지원하는 클래스 만들기

```
h1 = House('h1', 'Cape')
h1.add_room(Room('Master Bedroom', 14, 21))
h1.add_room(Room('Living Room', 18, 20))
h1.add_room(Room('Kitchen', 12, 16))
h1.add_room(Room('Office', 12, 12))

h2 = House('h2', 'Ranch')
h2.add_room(Room('Master Bedroom', 14, 21))
h2.add_room(Room('Living Room', 18, 20))
h2.add_room(Room('Kitchen', 12, 16))

h3 = House('h3', 'Split')
h3.add_room(Room('Master Bedroom', 14, 21))
h3.add_room(Room('Living Room', 18, 20))
h3.add_room(Room('Office', 12, 16))
h3.add_room(Room('Kitchen', 15, 17))
houses = [h1, h2, h3]
```

비교 연산을 지원하는 클래스 만들기

```
print("Is h1 bigger than h2?", h1 > h2)
print("Is h2 smaller than h3?", h2 < h3)
print("Is h2 greater than or equal to h1?", h2 >= h1)
print("Which one is biggest?", max(houses))
print("Which is smallest?", min(houses))
```

```
Is h1 bigger than h2? True
Is h2 smaller than h3? True
Is h2 greater than or equal to h1? False
Which one is biggest? h3: 1101 square foot Split
Which is smallest? h2: 846 square foot Ranch
```

"표준 비교 연산자(>=,!=,<= 등)를 사용해 클래스 인스턴스를 비교하고 싶다. 하지만 특별 메소드를 너무 많이 작성하고 싶지는 않다."

"이때 functools.total_ordering 데코레이터를 사용하면 과정을 단순화할 수 있다. 클래스에 데코레이터를 붙이고 __eq__()와 비교 메소드 (__lt__,__le__,__gt__,__ge__)중 하나만 더 정의하면 된다."